Embedded Systems 1: Hardware Description Languages (HDLs) for Verification

Davide Zoni PhD
email: davide.zoni@polimi.it
webpage: home.deib.polimi.it/zoni
Outline

• How to test an RTL design
  ➔ Verification vs Validation
  ➔ Simulation-based Verification
  ➔ Formal Verification
  ➔ Assertion-based Verification

• Verilog for Simulation-based Verification
  ➔ System tasks: $display, $monitor, $write,$display, $finish
  ➔ Procedural Blocks: initial blocks, always blocks
  ➔ Simulation Delay Models

• Tools: Simulation, Synthesis, Implementation
  ➔ Xilinx Vivado Overview

• Hands-on and Examples
  ➔ Combinatorial, Sequential, FSM
  ➔ Embedded Processor: An OpenRisc1000 Implementation
References

• Material on the website


• “Hardware Verification with System Verilog”, Mike Mintz and Robert Ekendahl, Springer 2007

NOTE:
The two classes related to “Introduction to Hardware Design and Verification with SystemVerilog” within the ES1 MSc course are intended as an off the ground presentation of the topic.
Several advances topics are left aside and not covered. These more advanced topics are not part of the written exam and will be introduced to those students involved with the ES1 project.
Recap: Essential Verilog for design

- Module is the basic building block for the hardware description
  - Input output interface (how to connect with the rest of the hierarchy)
  - Body (description of the semantic)
- Everything is parallel
  - Always blocks and continuous assignments (their order is irrelevant)
    - NOTE: within an always block order of the statements matters!
- Different types of assignments
  - Non-procedural assignments (Outside the always block)
    - Continuous assignments (keyword `assign`)
  - Procedural assignments (Only within an always block)
    - Non-blocking assignments (`<=`)
    - Blocking assignments (`=`)
- Two types to represents the information
  - Wire (net type), reg (variable type) in the form of bit vectors
  - 4-value logic (0,1,X,Z)
Verification VS Validation

- **Verification**: set of techniques to check if the design satisfies the specification. Are we correctly building the system? Are we building the system right?
  - Internal process: from the early design stages till post-implementation
  - It has to comply with a set of specifications that can eventually be ambiguous or incomplete, then leading to errors/bugs

- **Validation**: it answers the question: is the design fulfill with the customer’s expectations? Are we building the right system?
  - It is an external process
  - The fact that the design is doing something does not mean it is satisfactory
Verification

- The goal of verification
  - Ensure correct timing and functional behavior for the Design Under Test (DUT)
  - Usually 50% / 70% of the whole design project
  - Verification team is usually twice as big as the design team

- Functional Verification
  - Simulation
  - Formal proof

- Timing Verification (NOT COVERED IN THIS CLASS)
  - Dynamic Timing Simulation (DTS)
  - Static Timing Analysis (STA)
Functional Verification

- Simulation-Based Verification
  - Direct
  - Random
  - Constrained Random

- Formal Verification
  - Model Checking
  - Equivalence Checking
  - Theorem Proving

- Assertion-Based Verification
  - Dynamic Formal Verification
Traditionally, the quality of the verification was measured with the help of code coverage tools. The code coverage reflects the “executed/visited” HDL code.

- **Structural/Code Coverage:**
  - This will give information about how many lines/statements of the HDL code are stressed, in terms of how many times they are “visited”.
  - This coverage is collected by the simulation tools
  - Almost useless to verify the semantic of the HDL description

- **Functional coverage:**
  - This coverage is user defined
  - User will define the coverage points for the functions to be covered in DUT
  - This is under user control
  - It is made of two parts: what to visit and how to generate test vectors to reach the modules/functionalities of the design to be tested.
**Coverage Analysis: Check Points**

**F1:** sample the generated tests. The coverage stats are updated even if the test is not actually sent to the DUT

**F2:** sample the input vector at the DUT input. Collect DUT input coverage information

**F3:** sample intermediate values within the DUT. Traditionally assertion-based techniques are used to test this portion of the design space

**F4:** sample the output vector at the DUT output. Collect DUT output coverage information
The (Simplified) Hardware Design Flow

**Flow**

1. **Verilog Code**
   - `input a,b;`
   - `output [1:0];`
   - `assign sum={1'b0,a}+{1'b0,b};`

2. **Post Synthesis Code**
   - `G1 “and” n1 n2 n5`
   - `G2 “and” n3 n4 n6`
   - `G3 “or” n5 n6 n7`

3. **Post Mapped Code**
   - `G1 “and_tlib” n1 n2 n5`
   - `G2 “and_tlib” n3 n4 n6`
   - `G3 “or_tlib” n5 n6 n7`

4. **Post Place & Route Code**
   - `Netlist + accurate propagation delays`
   - `Ready to go!`

**Possible Actions**

- **Behavioral simulation**
  - Semantic and syntax checks
  - No physical net/gate propagation delays

- **Post-synthesis simulation**
  - Semantic of the synthesized code
  - No physical net/gate propagation delays

- **Post-mapped simulation**
  - Semantic of the synthesized code
  - Physical gate propagation delays
  - No net propagation delays

- **Post-place and route simulation**
  - Semantic of the synthesized code
  - Physical gate propagation delays
  - No net propagation delays
The Simulation Flow (Verilog 2001)

From previous time slot

- **Blocking assignments**
  - Any order between different always blocks
  - Fixed order within begin/end in the same always
  - Evaluate Right Hand Side (RHS) of Non-blocking assignments (NBAs)
- **Continuous assignments**
- **$display**

### Current time slot

- Active
- Inactive
- NBA
- Postponed

- #0 blocking assignments (do not use #0 delays. Most of the time they are useless and the same semantic can be modeled with a different description)
- Update Left Hand Side of Non-blocking assignments (NBAs)
- $monitor, $strobe system tasks

To next time slot

NOTE: No fixed evaluation order

The testbench is the infrastructure for testing. It is made of 3 parts:

- **design under test (dut)**: the RTL design that you want to test
- **tb_signal_generator**: it generates the stimuli for the dut.
  - [It contains the semantic of the test!!]
- **tb_monitor**: checks the output of the dut to report semantic/functional errors
The `testbench` module

`timescale [timeunit]/[timeprecision]

module minimal_tb_template();

always #5 clk=~clk; //clock generation

reg in1,in2; wire out1; //in/out to the dut

initial //initial block (even more than one)
begin
    clk <= 0; //init clk
    /*init var + procedures to generate
       input signals to dut */
end

//dut instance
dut #(parameter .P1(...), ...)  
d0 (.in1(in1), .in2(in2), .out1(out1));

//tb_monitor to check the dut output
endmodule

- **timescale** directive specifies the unit of time and the precision to be used during the simulation
  - Example: `timescale 1ns/10ps
  - NOTE: Inherited from the last processed module if not specified in the .v file

- The `testbench` module has no primary inputs and outputs, but can have parameters

- Verilog comments have the same C/C++ syntax
  - // comment until the end of the line
  - /* multi-line comment */

- Input to the **dut** are **reg**
- Output from the **dut** are **wire**
- The **initial block** is a procedural block that is executed once at the simulation startup
  - Not synthesizable
Clock, Reset and Maintainability
Behavioral Verilog: Clock Signal

- The clock signal is made of non-synthesizable Verilog statements, i.e. delays

- [RULE of THUMB] Use blocking assignments to update the clock [in a synchronous design]:
  - Avoid races with other signals that are updated in the same time-step
  - Eventually use non-blocking assignments for all the other signals
  - The use of non-blocking clock update assignment is possible even if it is considered “improper”
Behavioral Verilog: Reset Signal

**CORRECT DESIGN**

- The clock \([clk]\) and the reset \([rst]\) are updated during the same time-step
  - We can observe simulation misbehaviors
  - **NOTE:** all the signals in the TB that are synchronous to the clock must be updated using NBAs.

```verilog
always #5 clk=~clk

initial begin //can race!!!
  clk<=0; rst=1;
  #5 rst=0;
end
```

**IMPROPER DESIGN**

- The clock \([clk]\) and the reset \([rst]\) are updated during the same time-step
  - We can observe simulation misbehaviors
  - **NOTE:** the description can be used ONLY if an asynchronous reset signal is used

```verilog
always #5 clk=~clk

initial begin //no races!!!
  clk<=0; rst=1;
  #5 rst<=0;
end
```
The `timescale directive: rounding and truncation errors

**Truncation**

```vhdl
`timescale 1ns/1ns
reg clk;
parameter clk_period=25;
always #(clk_period/2) clk=~clk;
```

**Rounding**

```vhdl
`timescale 1ns/1ns
reg clk;
parameter clk_period=25;
always #(clk_period/2.0) clk=~clk;
```

**Correct Implementation**

```vhdl
`timescale 1ns/100ps
reg clk;
parameter clk_period=25;
always #(clk_period/2.0) clk=~clk;
```

- The design is simulated using an event driven simulation approach
  - **timeunit**: the unit of time for any kind of delay
  - **timeprecision**: how many events the simulator can generate at most between two timeunits. Also named **time steps**

- **timeunit** always bigger than **timeprecision**
  - They can be equal, while rounding and truncation errors can arise

- Different simulators allow you to specify both **timeunit** and **timeprecision** directives without cluttering the hardware source files
  - Rounding errors: due to the precision of the used computing machine
  - Truncation errors: due to the used arithmetic
Behavioral Verilog: Maintainability 1

**IMPROPER DESIGN**
- Use `parameter` and `define` to increase the reusability and maintainability of your testbenches
  - `parameter clk_period`
  - `define CLK_PERIOD`

**CORRECT DESIGN**
- Parameters are better than defines:
  - parameters are properties of the module
  - defines are just placeholders
parameter clk_period = 10;
parameter rst_delay=10;
...
always #(clk_period/2) clk=~clk

initial begin //no races!!!
  clk<=0; rst=1;
  #(rst_delay) rst<=0;
end

---

**IMPROPER DESIGN**

- Use the clk to synchronize the update of your design synchronous inputs
  
  → NOTE: maybe not the best choice if the reset is asynchronous with respect to the clock!!

---

**CORRECT DESIGN**

- The reset (rst) signal is clear after n_cycle of the clock (clk)
  
  → @(posedge clk) is a non-synthesizable delay construct
  
  → @(posedge clk) doesn’t mean anything from the physical viewpoint
Delay Models in Verilog
Verilog allows to model delays in the hardware description as well as in the testbenches for behavioral simulations

- **Inertial delay model**: the inertia of a circuit node to change value. It abstractly models the RC circuit seen by the node
  - ALTERNATIVE DEFINITION: measure of the input “hold time” to get a change in the output.
  - NOTE: A pulse of duration less than the inertial delay does not contain enough energy to cause the device to switch

- **Transport delay model**: the propagation time of signals from module inputs to its outputs
  - NOTE: Time taken by the signal to propagate through the net, also known as time of flight

```
Input                        
Output (inertial delay)      
Output (transport delay)     
```

## Behavioral Verilog Delay Models: Blocking Assignments (BA)

Adding RHS or LHS delays to model combinatorial logic is common while flawed:
- The behavioral delay is not synthesizable: it does not have a physical counterpart
- Depends on the timescale directive
- Functional Implications

<table>
<thead>
<tr>
<th>always@(a)</th>
<th>y = ~a</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>No delay</strong></td>
<td>• the delay is imposed by the gate from the technology library used to map the NB statement (post synthesis)</td>
</tr>
<tr>
<td></td>
<td>➔ Correct Implementation for any synthesizable Verilog description</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>always@(a)</th>
<th>y = ~a</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Left-hand-side delay</strong></td>
<td>• Evaluate Right-Hand-Side (RHS) then update the Left-Hand-Side (LHS) after 5 timeunit. The time of the LHS update is independent from any further change in the RHS</td>
</tr>
<tr>
<td></td>
<td>➔ What if an update happens at 2.5 timeunits?</td>
</tr>
<tr>
<td></td>
<td>➔ The first value is lost (single LHS update event is created)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>always@(a)</th>
<th>y = #5 ~a</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Right-hand-side delay</strong></td>
<td>• Immediately evaluate the RHS, then wait 5 timeunit before updating the LHS of the statement</td>
</tr>
<tr>
<td></td>
<td>➔ What if an update happens at 2.5 timeunits?</td>
</tr>
<tr>
<td></td>
<td>➔ The first value is kept, since RHS is immediately evaluated (single LHS update event is created)</td>
</tr>
</tbody>
</table>

- **Do not use LHS or RHS delays in synthesizable combinatorial logic**
- **Use LHS only for testbench statements to separate series of stimuli**
Adding RHS or LHS delays is also possible for NBAs. While the same implications discussed for BAs are still valid, the RHS delay can be used to model the transport delay in combinatorial logic. Still not synthesizable!

### Summary

- **Always@ (a)**
  - **y <= ~a**
    - No delay
    - (correct)
  - **Always@ (a)**
    - #5 y <= ~a
      - Left-hand-side delay
      - (HW nonsense)
  - **Always@ (a)**
    - y <= #5 ~a
      - Right-hand-side delay
      - (correct for transport delay modeling)

- **Correct Implementation for any synthesizable Verilog description**
  - Evaluate Right-Hand-Side (RHS) then update the Left-Hand-Side (LHS) after 5 timeunit
    - What if an update happens at 2.5timeunits?
    - The first value is lost!!
    - NBA are less efficient to simulate, thus LHS delay for NBA is also discouraged for testbenches.
  - Immediately evaluate the RHS, then wait 5 timeunit before updating the LHS of the statement
    - What if an update happens at 2.5timeunits?
    - The first value is kept and the following are queued. (multiple LHS update events are generated)

- **Do not use LHS delays to model combinatorial logic using NBA (HW nonsense )**
- **Use RHS delays to model the transport delay in combinatorial logic using NBA (new output is queued)**
Adding RHS or LHS delays to continuous assignments is also possible.

- It makes them not synthesizable
- Used to model the inertial delay in combinatorial logic to prevent undesired glitches

---

**assign y = ~a**

No delay *(correct)*

- the delay is imposed by the gate from the technology library used to map the statement (post synthesis)

**assign #5 y = ~a**

Left-hand-side delay *(correct for inertial delay modeling)*

- LHS delay in continuous assignment models the inertial delay of the logic
  - If multiple event updates a within the delay period, only the generated update event is kept
  - NOTE: continuous assignments do not queue new values, while keep the last within the #5 delay to model the inertial delay

**assign y = #5 ~a**

Right-hand-side delay *(illegal syntax)*

- Question: why don’t BAs model the inertial delay?
  - Hint: LHS delay with continuous assignments reschedule the event in the future, if any, while LHS delay with BA just update the LHS value
Design Under Test (DUT)
Observability: System tasks
$monitor, $display, $write, $strobe

Unsynthesizable system tasks to print out useful information on the simulated design.

- **$strobe**: print the values at the end of the current timestep.
  - Executed in the postponed region
  - Example: $strobe("%t %b", time, signal);

- **$display**: print the immediate values
  - Executed in the active region
  - Example: $display("%t %b", time, signal);

- **$monitor**: print the values at the end of the current timestep if any values changed.
  - $monitor can only be called once; sequential calls will override the previous
  - Executed in the postponed region
  - Example: $monitor("%t %b", $time, signal);

- **$write**: same as $display but doesn't terminate with a newline (\n)
  - Executed in the active region
  - Example: $write("%t %b\n", time, signal);
The simulation time

Verilog allows to report the simulation time by accessing to specific system variables

- The time value is always scaled to the timeunit
- The time value can be stored in a variable of proper size if not directly printed

Examples

```verilog
real time=$realtime;
integer=$stime;
integer=$time;

$display("%t",$time);
```

- `$realtime` returns a 64-bit floating point value scaled to the time precision
- `$stime` return a 32-bit unsigned integer value
  - If current value is bigger than 32-bit, the lower 32-bit only are returned
- `$time` return 64-bit integer value scaled to the timeunit, i.e., truncation of fractional timeunit delays
- The `$timeformat`(<units>, <precision>, <suffix>,<minimum field width>) allows to specify the `$time`, `$realtime`, `$stime` return value
  - If not specified the default values:
    - `<units>`:
    - `<precision>`: 0
    - `<suffix>`: null string
    - `<minimum field width>`:
Design Under Test (DUT) Observability with VCD
Value Change Dump (VCD) File Extraction

The Value Change Dump file can store all the transitions for all the signals in the design during the entire simulation.

- Verilog offers system tasks to instruct the simulator on the vcd dump procedure.
- Several simulators allow a specific, tool-dependant solution for the VCD file dump without affecting the .v files.
- The VCD can be read using a waveform visualizer:
  - Gtkwave [opensource]
  - Simvision [cadence]
  - Xsim waveform [xilinx, free*]

From within the testbench .v file:

```verilog
$dumpfile("file.vcd");
$dumpvars(level,<module_instance/variable>);
```

- 0 dump all variable in the module and all of its submodules
- 1 dump all variables in the module
- `<level>` dump all variables in the module and all the variables in the children up to level = `<level>`-1